CS61A Week | Midterm 3 Review (v1.0)

Basic Info

Your login:
Your section number:
Your TA's name:
Your signature:

Midterm 3 is going to be held on Wednesday 7-9p, at 2050 VLSB. There is going to be a group question before the individual exam.

Box and Pointer Diagrams

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a box and pointer diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram first!

1.
```
(let ((x (list 1 2 3 4)))
       (set-cdr! (cddr x) (car x))
       x)
```

2.
```
(let ((x (list 1 2 3 4)))
       (set-car! (cddr x) (cddddr x))
       x)
```

3.
```
(let ((x (list 1 2 3 4)))
       (set-car! (cddr x) x)
       x)
```

4.
```
(let ((x (list 1 2 3)))
       (set-car! x (cdr x))
       (set-cdr! (car x) 5)
       x)
```

5.
```
(let ((x (list 1 2 3)))
       (set-car! x (list 'a 'b 'c))
       (set-car! (cdar x) 'd)
       x)
```

Object Oriented Programming

1. Question 2 of MT3, Fall 2001

```
(define-class (scoop flavor)
  ; maybe (parent (cone)) - see part (a) below
  )

(define-class (vanilla)
  (parent (scoop 'vanilla)))
(define-class (chocolate)
  (parent (scoop 'chocolate)))

(define-class (cone)
  ; maybe (parent (scoop)) - see part (b) below
  (instance-vars (scoops '()))
  (method (add-scoop new)
    (set! scoops (cons new scoops)))
  (method (flavors)
    (map (*see part b below*) scoops)))
```

   (a) Which of the parent clauses shown above should be used?
   The `scoop` class should have `(parent (cone))`
   The `cone` class should have `(parent (scoop))`
   Both
   Neither

   (b) What is the missing expression in the flavors method?

   (c) Which one of the following is the correct way to add a scoop of vanilla ice cream to a cone named my-cone?
   `(ask my-cone 'add-scoop 'vanilla)`
   `(ask my-cone 'add-scoop vanilla)`
   `(ask my-cone 'add-scoop (instantiate 'vanilla))`
   `(ask my-cone 'add-scoop (instantiate vanilla))`

2. Question 3 of MT3, Fall 1998

   We are going to prepare a simulation of an FM car radio. To simplify the problem we'll restrict our attention to tuning, not to volume or balance or anything else a radio does. This radio features digital tuning. There are six buttons that can be preset to particular stations; for manual tuning, there are up and down buttons that move to the next higher or lower frequency. (FM frequencies are measured in megahertz and have values separated by 0.2: 88.1, 88.3, 88.5, 88.7, 88.9, 90.1, etc.) To simplify the problem further, we'll ignore the boundary problem of what to do when you're at the lowest assigned FM frequency and try to go down below that frequency. Just pretend you can keep going up or down forever.

   Use the OOP language (define-class and so on).

   (a) Create a button object class that accepts these messages:
   `set-freq! 93.3` sets the button's remembered frequency
   `freq` returns the remembered frequency
   The initial frequency should be zero (because the buttons don't have settings initially).

   (b) Create a radio object class that has six buttons, numbered 0 through 5, and accepts these messages:
   `set-button! 3` sets button 3 to the radio's current frequency
   `push 3` sets the radio to button 3's frequency
   `up` sets the radio to the next higher frequency down sets the radio to the next
   `down` frequency freq returns the radio's current frequency
   The radio's initial frequency should be 90.7 MHz. Points to remember: Your radio has to use six of your button objects; you needn't check for invalid argument values in the methods. Hint: Give your radio a list of six buttons, and use list-ref to get the one you want.

Environment Diagram

1. Question 6 of Final, Fall 1997

```
(define (kons a b)
  (lambda (m)
    (if (eq? m 'kar) a b)))
(define p (kons (kons 1 2) 3))
```

2. Question 9 of Final, Fall 1998

```
(define x 3)
(define y 4)
(define foo
    ((lambda (x) (lambda (y) (+ x y)))
     (+ x y)))
(foo 10)
```

List Mutations

1. Question 2 of MT3, Fall 1994

   Write `make-alist!`, a procedure that takes as its argument a list of alternating keys and values, like this:

   ```
   (color orange zip 94720 name wakko)
   ```

   and changes it, by mutation, into an association list, like this:

   ```
   ((color . orange) (zip . 94720) (name . wakko))
   ```

   You may assume that the argument list has an even number of elements. The result of your procedure requires exactly as many pairs as the argument, so you will work by rearranging the pairs in the argument itself. Do not allocate any new pairs in your solution!

2. Question 2 of MT3, Spring 1996

   Write `list-rotate!` which takes two arguments, a nonnegative integer n and a list seq. It returns a mutated version of the argument list, in which the first n elements are moved to the end of the list, like this:

   ```
   > (list-rotate! 3 (list 'a 'b 'c 'd 'e 'f 'g))
   (d e f g a b c)
   ```

   You may assume that $0 \leq$ `n < (length seq)` without error checking.

   Note: Do not allocate any new pairs in your solution. Rearrange the existing pairs.

Vector

1. You've seen vectors. You've seen tables. We want to implement tables with vectors. To make things easier, we're going to assume that there are two vector tables that manage keys and values. Assume that the corresponding key/value pair has the same index.

   Write `vector-lookup` for vector-tables. It acts just like lookup in tables. It takes a key and returns the corresponding value if the key is in the table and `#f` otherwise.

2. Write `vector-reverse!` that does the obvious thing. Do not create new vectors!

   Hint: You might want to write a helper called `vector-swap!` that takes in a vector and two indices and swap the elements at those indices.

Concurrency

1. What are the possible values of x after the following is executed:

```
(define x 10)
(parallel-execute (lambda() (set! x (+ 5 x)) (set! x (* x 3)))
                  (lambda() (if (> x 16)
                                (set! x 100)
                                (set! x (- x 20)))))
```

2. Question 13 of Final, Spring 2003

   Given the following definitions:

```
(define s (make-serializer))
(define t (make-serializer))
(define x 10)
(define (f) (set! x (+ x 3)))
(define (g) (set! x (* x 2)))
```

   Can the following expressions produce an incorrect result, a deadlock, or neither? (By "incorrect result" we mean a result that is not consistent with some sequential ordering of the processes.)

   (a) `(parallel-execute (s f) (t g))`

   (b) `(parallel-execute (s f) (s g))`

   (c) `(parallel-execute (s (t f)) (t g))`

   (d) `(parallel-execute (s (t f)) (s g))`

   (e) `(parallel-execute (s (t f)) (t (s g)))`